

# Máster Avanzado de Programación en Python para Hacking, Big Data y Machine Learning

BUENAS PRÁCTICAS DE  
PROGRAMACIÓN EN PYTHON

# LECCIÓN 01

## Control de errores, pruebas y validación de datos

# ÍNDICE

Introducción

Objetivos

Control de errores

Pruebas y validación de datos

Conclusiones

# INTRODUCCIÓN

- En esta lección aprenderemos a crear códigos de programación en los que utilizaremos control de errores y pruebas para desarrollar códigos más eficientes e interpretables.
- Estudiaremos la importancia de realizar pruebas y validación en los datos que utilizamos como entrada en nuestros programas.
- Validar y verificar que nuestro programa responde favorablemente ante cualquier entrada aumenta la calidad de nuestro software.

# OBJETIVOS

Al finalizar esta lección serás capaz de:

- 1 Conocer la importancia del control de errores
- 2 Saber gestionar y depurar los errores de un código de programación.
- 3 Conocer el concepto de programación defensiva y cómo aplicarlo en nuestro día a día.
- 4 Conocer los diferentes mecanismos y herramientas disponibles para validar la entrada de datos en nuestro código Python.

## 1. CONTROL DE ERRORES

Hasta ahora, hemos utilizado los mensajes de error para interpretar y tratar de entender qué está ocurriendo en nuestro código.

Conocer e identificar el error nos ayudará a buscar remedio.

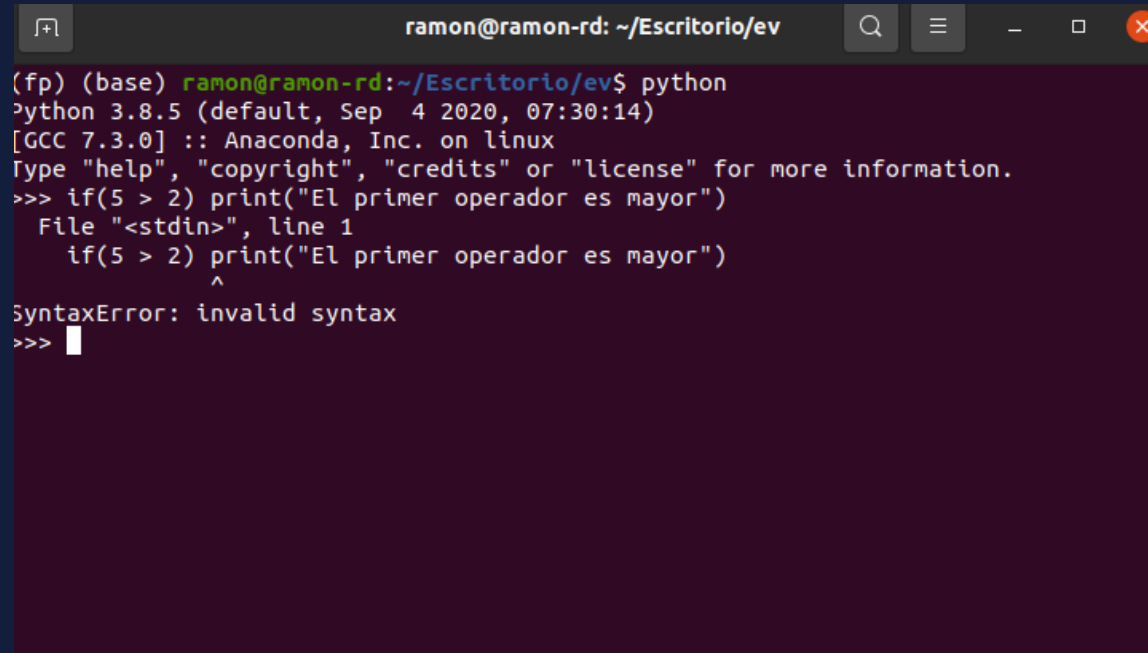
Distinguimos entre dos tipos de errores:

- Errores de sintaxis.
- Errores lógicos.

## 1.1. Errores de sintaxis

Tipo de error más común.

Muestran un mensaje donde indican que o bien no se ha escrito correctamente un comando de Python o que por algún motivo nos estamos “saltando” las reglas.

A screenshot of a terminal window titled 'ramon@ramon-rd: ~/Escritorio/ev'. The terminal shows the output of running 'python' in a JupyterLab environment. It displays the Python version (3.8.5), GCC version (7.3.0), and Anaconda information. The user enters a Python command: 'if(5 > 2) print("El primer operador es mayor")'. The terminal shows the file '<stdin>', line 1, and the code 'if(5 > 2) print("El primer operador es mayor")'. A caret points to the opening parenthesis of the if statement, and the error message 'SyntaxError: invalid syntax' is displayed. The prompt '>>>' is shown at the end of the line.

```
(fp) (base) ramon@ramon-rd:~/Escritorio/ev$ python
Python 3.8.5 (default, Sep  4 2020, 07:30:14)
[GCC 7.3.0] :: Anaconda, Inc. on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> if(5 > 2) print("El primer operador es mayor")
      File "<stdin>", line 1
        if(5 > 2) print("El primer operador es mayor")
            ^
SyntaxError: invalid syntax
>>> 
```

## 1.1. Errores de sintaxis

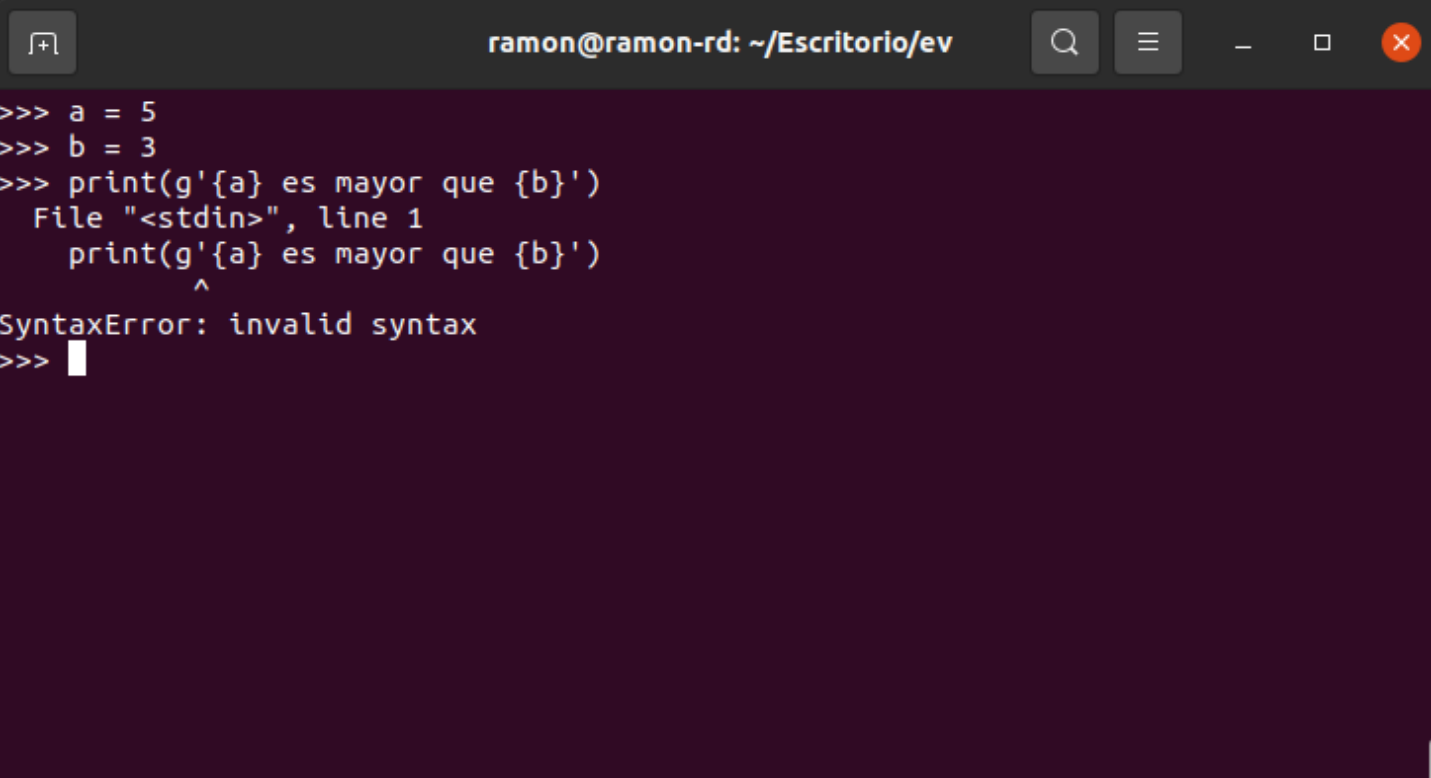
El analizador sintáctico de Python repite la línea de programación donde ha encontrado el error y muestra una pequeña flecha que apunta al último punto de la línea en la que se encontró el error.

Nótese que el analizador sintáctico también muestra la línea donde se ha producido el error, con el objetivo de facilitar la tarea al programador a la hora de solucionar el problema.

Siguiendo esta línea, a continuación se muestra una captura con un error de sintaxis en un código de programación. ¿Podrías adivinar a qué se debe el error?



## 1.1. Errores de sintaxis

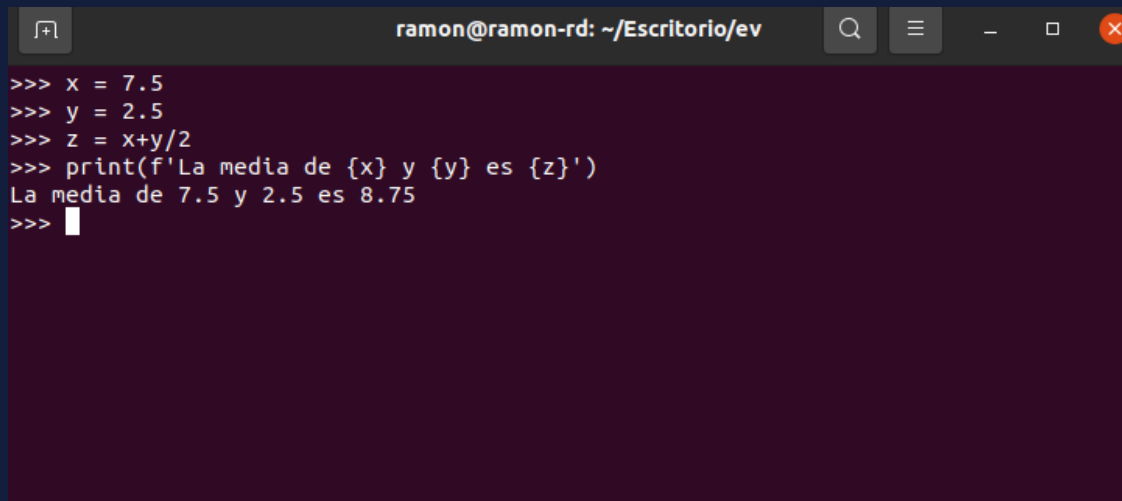


```
ramon@ramon-rd: ~/Escritorio/ev
>>> a = 5
>>> b = 3
>>> print(g'{a} es mayor que {b}')
File "<stdin>", line 1
    print(g'{a} es mayor que {b}')
          ^
SyntaxError: invalid syntax
>>> 
```

## 1.2. Errores lógicos

Son los más difíciles de detectar ya que darán resultados impredecibles o incluso podría llegar a bloquear el programa.

No obstante, este tipo de errores son muy fáciles de solucionar ya que podemos hacer uso de un **depurador** que ayudará a solucionar este tipo de problemas.



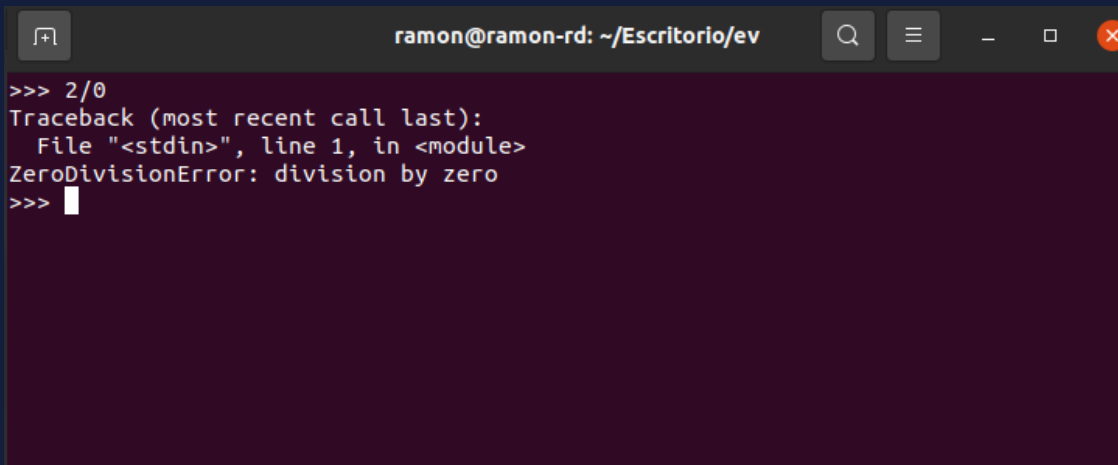
```
ramon@ramon-rd: ~/Escritorio/ev
>>> x = 7.5
>>> y = 2.5
>>> z = x+y/2
>>> print(f'La media de {x} y {y} es {z}')
La media de 7.5 y 2.5 es 8.75
>>>
```

## 1.3. Excepciones

(...) Incluso si un programa es sintácticamente correcto, puede producirse un error cuando se intenta ejecutar.

Los errores detectados durante la ejecución se denominan excepciones.

La mayoría de las excepciones no son manejadas por los programas y dan como resultado mensajes de error.

A screenshot of a terminal window with a dark background. The window title bar shows 'ramon@ramon-rd: ~/Escritorio/ev'. The terminal content shows a Python prompt '>>>' followed by the input '2/0'. This results in a 'Traceback (most recent call last):' message, followed by 'File "<stdin>", line 1, in <module>' and 'ZeroDivisionError: division by zero'. The prompt '>>>' is shown again with a cursor at the end.

```
ramon@ramon-rd: ~/Escritorio/ev
>>> 2/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> █
```

## 1.3. Excepciones

```
ramon@ramon-rd: ~/Escritorio/ev
>>> 7+numero*5
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'numero' is not defined
>>> 
```

```
ramon@ramon-rd: ~/Escritorio/ev
>>> '8'+2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
>>> 
```

### 1.3.1. Manejo de excepciones

Es posible desarrollar un programa capaz de manejar un conjunto de excepciones.

Las excepciones en Python pueden ser gestionadas por medio de las órdenes `try` y `except`.

Si está desarrollando un código en el que cree que podrían surgir excepciones, puede “proteger” el programa incluyendo dichas órdenes.

De este modo, si ocurre una excepción en su programa, en lugar de detenerse mostrando un error, le mostrará un mensaje de que ha ocurrido la excepción pero tiene la posibilidad de continuar con la ejecución del programa hasta el final.

### 1.3.1. Manejo de excepciones

La sintaxis es la siguiente:

```
try:
    operaciones deseadas
    ...
except ExceptionI:
    Si ocurre la primera excepción, ejecuta este bloque
except ExceptionII:
    Si ocurre la segunda excepción, ejecuta este bloque
    ....
else:
    Si no hay ninguna excepción, haz esto otro.
```

### 1.3.1. Manejo de excepciones

Vamos a un ejemplo práctico...

### 1.3.2. Excepciones definidas por el usuario

Una opción muy interesante que ofrece el lenguaje de programación Python es la posibilidad de crear excepciones personalizadas.

Las excepciones que podemos crear derivan de la clase Exception directa o indirectamente.

Este tipo de clases se pueden implementar como cualquier otro tipo de clases, pero en la práctica se tratan de desarrollar de la forma más simple y directa posible.

Muchas implementaciones declaran una clase principal y el resto de clases derivan de la clase base o principal. Veamos un ejemplo...



## 2. Pruebas y validación de datos

La validación de datos es una técnica que permite asegurar que los valores de entrada con los que vaya a operar un programa o función de Python están dentro de un determinado dominio.

Este tipo de técnicas se han convertido en una herramienta imprescindible de todo programador por varios motivos:

- Ayuda a prevenir errores de ejecución derivados de utilizar datos incompatibles o fuera de dominio.
- Aumenta la interpretabilidad del código.

## 2. Pruebas y validación de datos

Formas de comprobar el dominio de un dato:

- Comprobando el contenido del dato: que una variable sea de un tipo particular.
- Que el dato tenga una característica determinada.

Debemos pensar una forma de actuar o responder si se “dispara” alguna alarma que indique que algo no va bien, ya que es también nuestro objetivo mostrar información de interés al usuario que le ayude a detectar y solucionar el error.

Utilizar excepciones (del sistema o creadas por nosotros mismos) son una solución elegante.

## 2. Pruebas y validación de datos

Al uso concienzudo de este tipo de técnicas se le conoce como **programación defensiva**.

La programación defensiva persigue la mejora del software y su código fuente atendiendo a tres criterios principales:

1. **Calidad.** Adoptar medidas de programación defensiva reduce el número de fallos del software.
2. **Interpretabilidad.** Mejora la comprensibilidad y legibilidad del código, a prueba de una auditoría de código.
3. **Prevención de errores.** Permite que el software desarrollado se comporte de forma impredecible a pesar de que el usuario o terceras personas realicen acciones inesperadas sobre nuestro código.

## 2. Pruebas y validación de datos

# ¡¡IMPORTANTE!!

Los errores de software (bugs) pueden ser potencialmente utilizados por hackers para una inyección de código, ataques de denegación de servicio u otro ataque.

## 2.1. Técnicas de programación defensiva

Existen diversas técnicas de programación defensiva que un desarrollador puede adoptar para mejorar la calidad de su código.

- **Revisiones de código fuente.** Una revisión de código se refiere a que alguien diferente al autor original del código realice una auditoría del mismo.
- **Pruebas de software.** Crear test unitarios para comprobar que un determinado programa cumple su función y devuelve valores apropiados para una determinada entrada.

## 2.1. Técnicas de programación defensiva

Existen diversas técnicas de programación defensiva que un desarrollador puede adoptar para mejorar la calidad de su código.

- **Reduce la complejidad del código fuente.** Evitar hacer implementaciones complejas. Realiza el código de la manera más sencilla posible para que resuelva su función.
- **Reutilización del código fuente.** Siempre que sea posible, reutiliza el código. De nuevo, la idea es evitar crear nuevos bugs.
- **Problemas de legado.** Antes de reutilizar código fuente antiguo, bibliotecas o APIs, éste debe ser analizado y estudiado en detalle, considerando si las funcionalidades externas que queremos incluir en nuestro código son aptas para ser reutilizadas o si son propensas a problemas de legado.

## 2.2. Validaciones

Mostraremos un conjunto de casos en los que resulta de interés desarrollar un conjunto de comprobaciones y validaciones.

No obstante, existe una multitud de casos en los que pueden desarrollarse sistemas de validación de código. Es por ello que el objetivo de esta lección es el de mostrar al estudiante en qué consiste las pruebas y validación de datos, cuáles son sus beneficios y algunos ejemplos prácticos.

Es misión del desarrollador (estudiante en este caso) determinar cuándo es necesario incluir validaciones en su código y actuar en consecuencia.

### 2.2.1. Comprobaciones por contenido

Se utilizan cuando queremos validar que los datos recibidos como entrada para realizar una determinada operación contienen información apropiada.

Estas comprobaciones no podremos realizarlas siempre, ya que en algunas ocasiones puede ser muy laborioso comprobar si la entrada es correcta.

En el siguiente ejemplo, se muestra un fragmento de código que calcula el factorial de un número. En este código utilizamos el comando `assert` para comprobar que el contenido de la variable de entrada es mayor o igual que 0.



### 2.2.1. Comprobaciones de valores de entrada del usuario

Imaginad que necesitamos desarrollar una función que solicite al usuario que introduzca por teclado un número entero.

El usuario, de forma involuntaria o maliciosa, podría insertar un valor diferente a un entero, por ejemplo una cadena de caracteres.

Para manejar este tipo de errores, podemos hacer uso de la función *input* combinado con el manejo de excepciones.

### 2.2.1. Comprobaciones por tipo

Otra forma de realizar las comprobaciones por tipo es mediante el uso de la función `type`.

Esta función recibe por parámetro un valor o una variable y devuelve su tipo. Podemos comprobar que el tipo de cualquier variable que hayamos definido previamente.

### 2.2.1. Comprobaciones por tamaño de cadena

En algunas aplicaciones resulta de interés incluir un mecanismo de validación para comprobar que una contraseña es correcta.

En este ejemplo práctico (y sencillo), desarrollamos un método de validación para comprobar que la contraseña ingresada por el usuario tiene un tamaño mínimo.



## CONCLUSIONES

1

Las herramientas disponibles para llevar a cabo el control de errores en nuestro código.

2

Cómo gestionar y depurar errores de un código de programación por medio del manejo de excepciones.

3

Cómo llevar a cabo el concepto de programación defensiva a nuestras implementaciones, creando software robusto y de mayor calidad.

MUCHAS GRACIAS POR SU ATENCIÓN



[rrueda@grupomainjobs.com](mailto:rrueda@grupomainjobs.com)



Ramón Rueda Delgado  
<https://www.linkedin.com/in/ramon-rueda/>



[twitter.com/eiposgrados](https://twitter.com/eiposgrados)



[facebook.com/eiposgrados](https://facebook.com/eiposgrados)



[instagram.com/eiposgrados](https://instagram.com/eiposgrados)